Stack ADT

Stack ADT

- A stack is a list with the restriction that insertions and deletions can be performed only at the end of the list (called the **top**)
- Fundamental operations:
  - Push
  - Pop
  - Top
- LIFO (last in first out)
- Can be implemented with linked list or array data structures
- All operations are $O(1)$

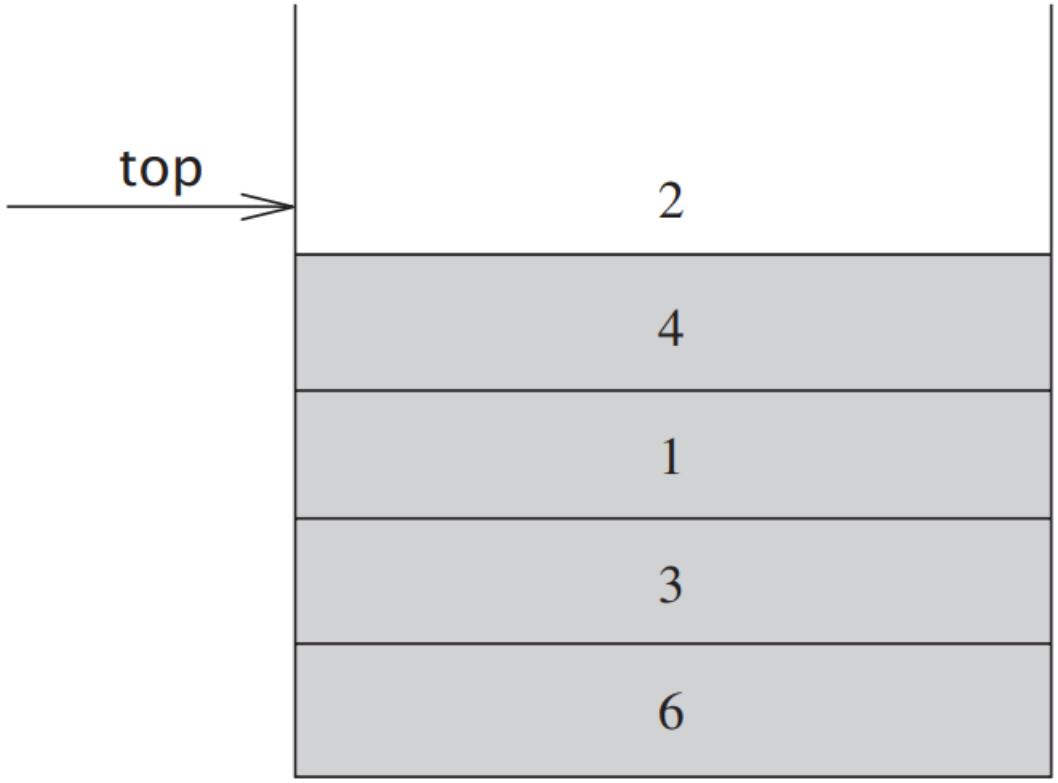


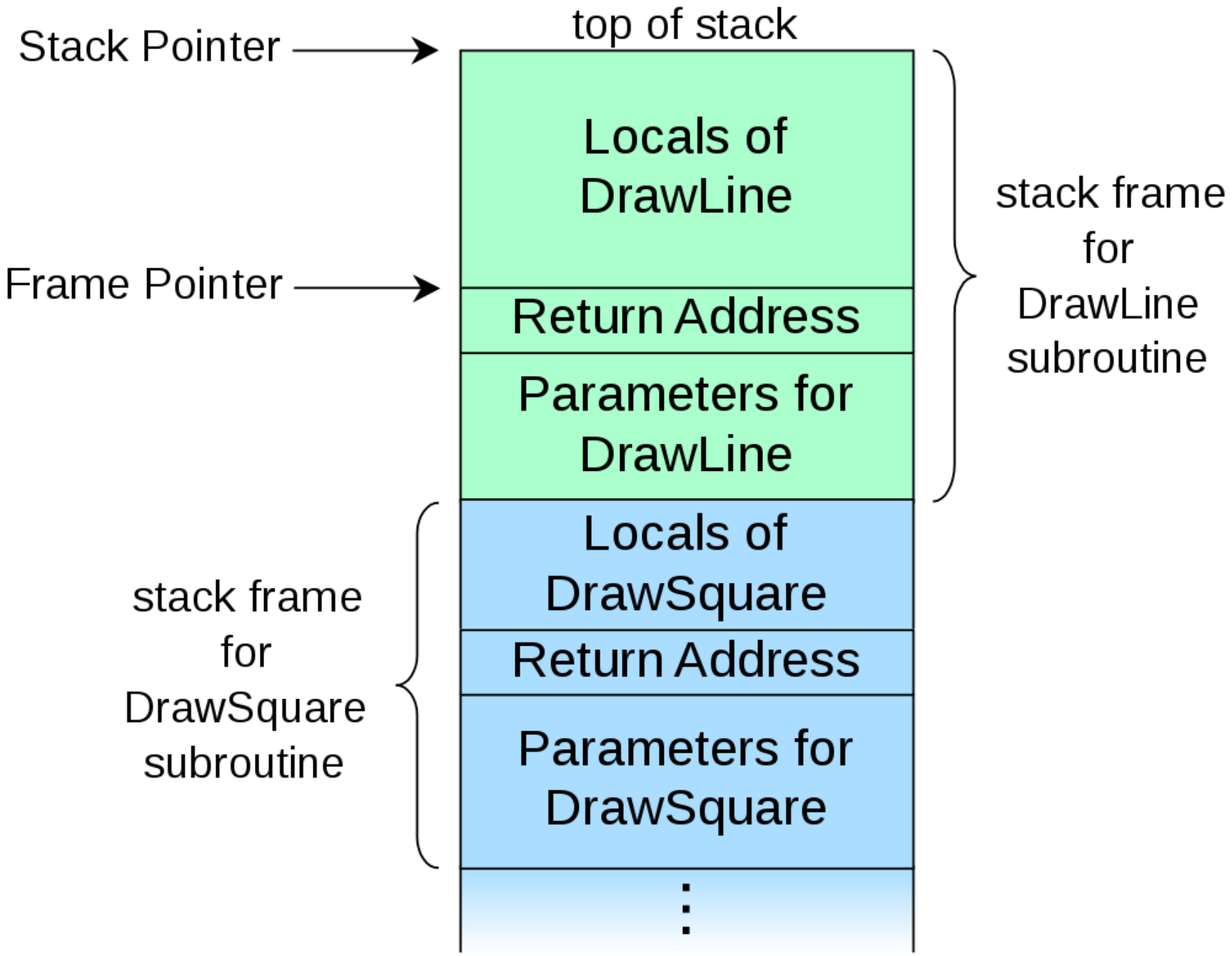
Figure 3.24 Stack model: Only the top element is accessible

Applications of stack

- Call stack (function calls)
- Balancing symbols (compilers)
- Evaluating and converting expressions
  - Assignment 2

Call stack

- A *call stack* is composed of *stack frames*
- each stack frame corresponds to a call to a subroutine which has not yet terminated with a return
- Stack frame at top of stack is for currently executing routine, includes (from top to bottom)
  - Local variables
  - Return address
  - Arguments to routine
- Unwinding
  - Returning from called function pops top frame off stack, leaving only return value



C++ automatic variables

- An ‘automatic variable’ is a *local variable* which is allocated and deallocated automatically when program flow enters and leaves the variable’s scope
- In Java, all objects are created using new, but compiler can optimize some to the stack instead of heap (see escape sequence analysis)

```
Stack<int> autoStack{}; // automatic variable (stack allocated)

Stack<int>* heapStack = new Stack<int>{}; // heap-allocated variable
```

Balancing symbols

- Compilers check programs for syntax errors
- Every right bracket or parenthesis must correspond to left counterpart
- Example: `[()]` is legal, but `[(])` is wrong
- Simple algorithm to check for balanced brackets:
  - Create empty stack
  - Read characters until end of file
  - If character is an opening symbol, push to stack
  - If character is a closing symbol and stack is empty, report error
  - If character is a closing symbol, pop stack
    - If symbol popped is not corresponding opening symbol, report error
  - At end of file, if stack not empty, report error

Symbol balancing example 1:

- Input sequence: [[( )]] { }
- S = empty
- S.push(‘[‘), S.push(‘[‘), S.push(‘(‘). Stack is now: [[(
- S.pop returns ‘(’ (ok), S.pop returns ‘[’ (ok), S.pop returns ‘[’ (ok)
- S.push(‘{‘). S.pop returns ‘{’ (ok).
- At end of sequence and stack is empty. Input is ok.

Symbol balancing example 2

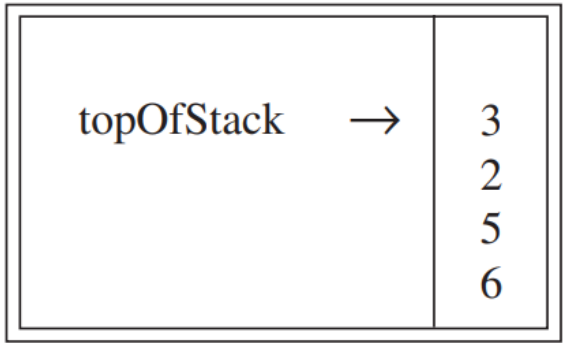
- Input sequence: { { [ ] } } (
- S = empty
- S.push('{'), S.push('{'), S.push('['). Stack is now { { [
- S.pop returns '[' (ok), S.pop returns '{' (ok), S.pop returns '{' (ok)
- S.push('(')
- At end of sequence but stack is not empty! Return error.

Infix and postfix expressions

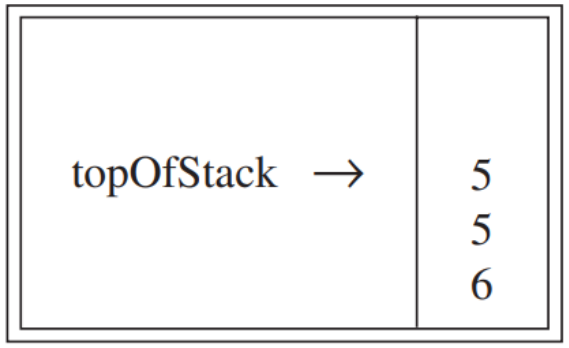
- Example expression: 4 + 5 + 6 * 2 = ?
- Depending on order of operations, could yield 21 or 30
- Can use brackets to change meaning of expression:
- 4 + 5 + (6 * 2) = 21 `expr1`
- (4 + 5 + 6) * 2 = 30 `expr2`
- Can write `expr2` above without brackets using *postfix expression:*
- 4 5 + 6 +2 * `expr2`
- 4 5 + 6 2 * + `expr1`
- Also called ‘reverse polish notation’

Example: evaluating postfix using stack

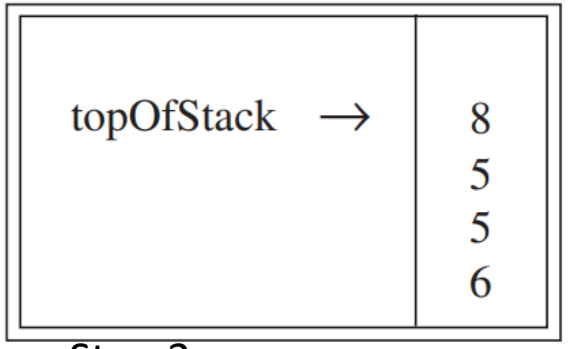
- Postfix expression: 6523+8*+3+*
- **Algorithm**: evaluate sequence from left to right. When we encounter a number, push it on the stack. When we encounter an operator, pop two elements off the stack, evaluate them using the operator, and push the result back on the stack. Time complexity is $O(N)$



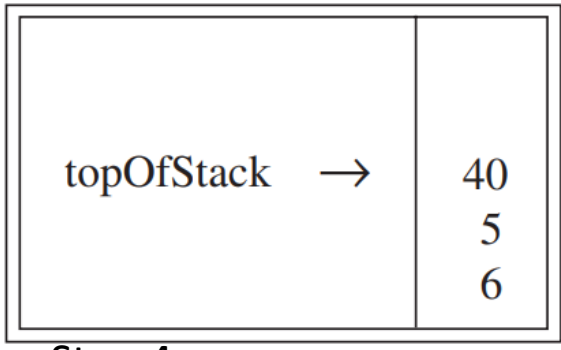
Step 1



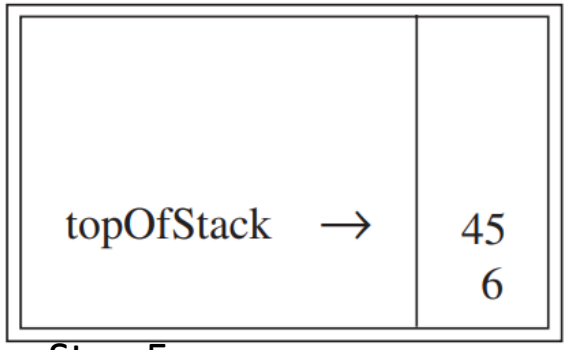
Step 2



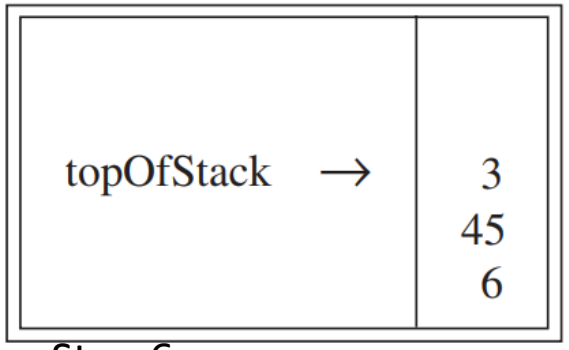
Step 3



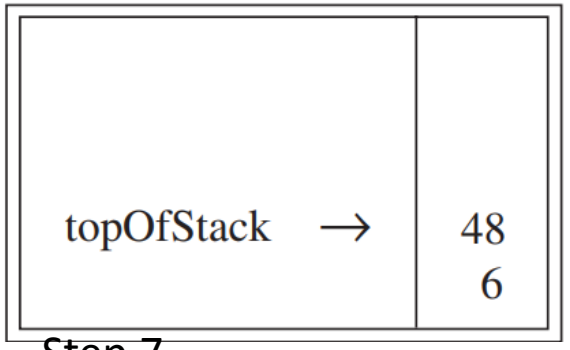
Step 4



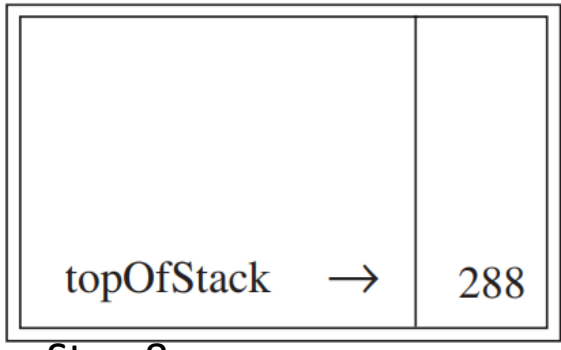
Step 5



Step 6



Step 7



Step 8

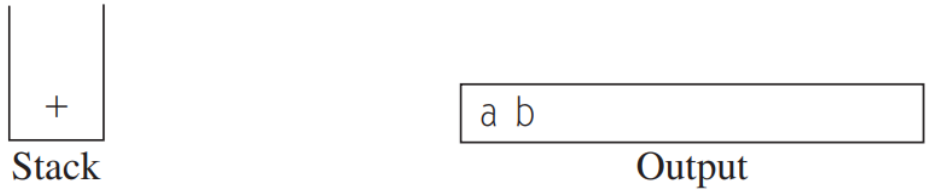
Infix to postfix conversion

- Stack can also be used to convert infix to postfix and vice-versa
- Example: **a+b*c+(d*e+f)*g** (infix) is equivalent to **abc*+de*f+g*+** (postfix)
- Algorithm: ($O(N)$ time complexity)
  - Initialize empty stack
  - Read infix expression from left to right
  - When operand encountered (a,b,c etc.) place it immediately in the output
  - When operator encountered (including *left* parenthesis), it depends (details below)
  - When *right* parenthesis encountered, pop operators off stack and write them to the output, until we encounter the left parenthesis, which is popped and discarded
  - When operator +, *, (encountered, pop entries from stack and write them to output until operator of lower priority is found (but only remove left parenthesis if current symbol is right parenthesis)
  - If at end of input (infix expression) pop all entries from stack and write to output

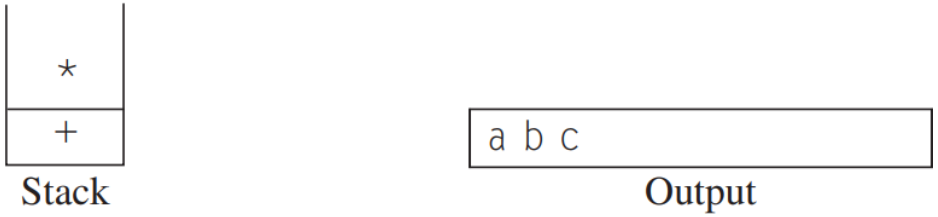
Infix to postfix conversion example

- Input: **a+b*c+(d*e+f)*g**

Step 1

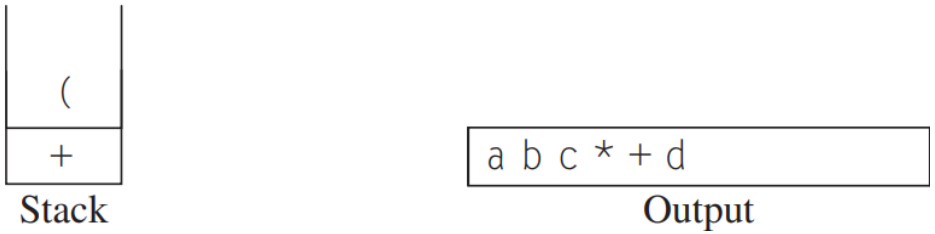


Step 2

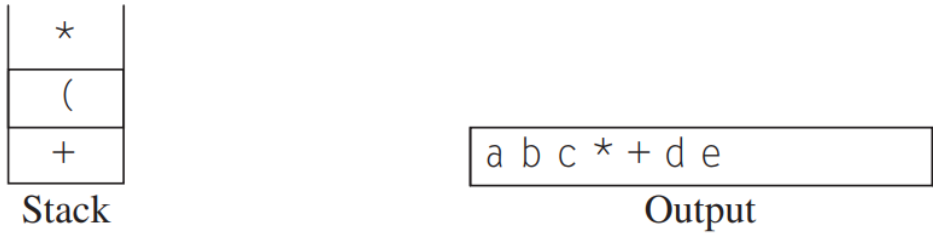


Step 3

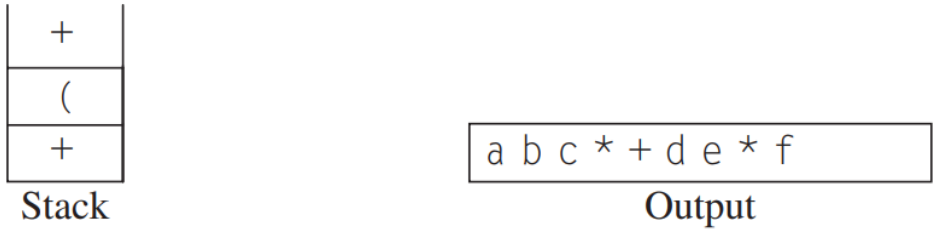
Step 4



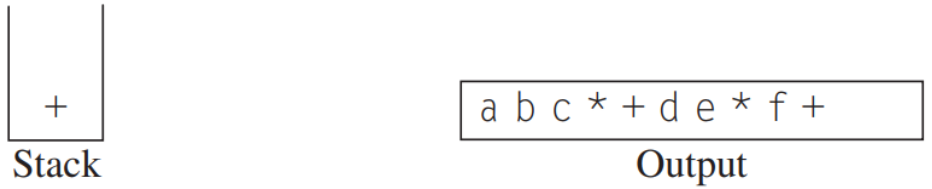
Step 5



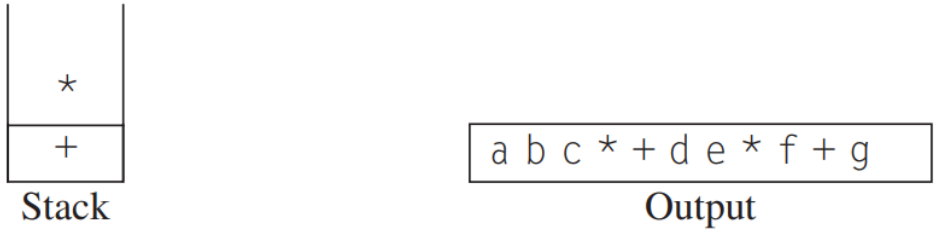
Step 6



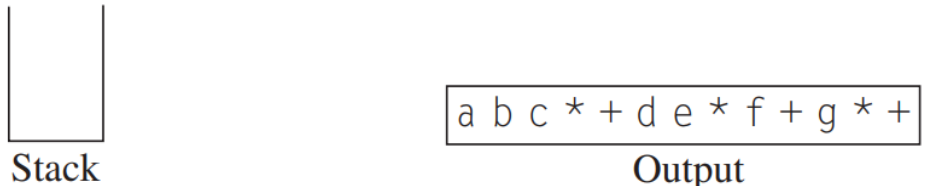
Step 7



Step 8



Step 9




stackoverflow